Answering XML Twig Queries with Automata

Bing Sun Bo Zhou Nan Tang Guoren Wang Ge Yu Fulin Jia
Northeastern University, Shenyang, China
{sunb,wanggr,yuge,dbgroup}@mail.neu.edu.cn

Abstract

XML is emerging as a *de facto* standard for information representation and data exchange over the web. Evaluation of twig queries, which allows users to find all occurrence of a multiple branch pattern in an XML database, is a core and complicate operation for XML query processing. Performance of conventional evaluation approaches based on structural join declines with the expansion of data size and query complexity. In this paper, a novel approach is proposed to compute twig queries with matching twig query automata and path schema trees. Moreover, we give the performance evaluation to demonstrate the high performance of our approach.

1 Introduction

Predicates are used to filter XML [18] node sets, which are a very important parts of location paths in XPath [19] and XQuery [20] specification. Simple predicates like value testing are common and easy to evaluate. Some predicates, nevertheless, are expressions containing paths which are rather complicated sometimes, and queries with these predicates are known as *twig* queries. It is a core operation in XML database and a hot spot in XML query processing field to find all XML nodes following a certain twig schema.

There is some research work on twig query evaluation adopting structural join [1, 6, 3, 13, 5, 16]. It simply breaks a query expression into basic steps, computes results of each, and joins them to get the final results. Although many optimization methods have been proposed, structural join still has some problems which infect its performance: Because it produces many sub-queries, the storage cost is very high and usually it cannot run in main memory. When computing join operations, all instances in each intermediary result must be accessed for test. It cannot directly compute some operators such as closure operators which are in regular path expression. Query rewriting which is a possible way to solve this problem has a very low efficiency. Moreover, it must use some schema information and statistic information of XML document while rewriting.

Automata techniques are widely used in XML query processing nowadays. Some XML stream processing systems use automata to represent user profiles such as XFilter [2] and YFilter [7], and use a SAX interfaced parser to parse XML documents and activate automata. However, in such systems, the roles of queries and data are reversed.

We have proposed an XML path query processing approach based on automata theory [15, 17, 25]. We call it basic automata match (AM) approach in this paper. Different from XFilter and YFilter, automata which represents the queries are dynamically generated rather than stored in the database. An index named path schema tree is used, which can make full use of schema information of the document and reduce the operation of accessing instance data to match the query automata in order to promote query performance.

In this paper, we extend basic AM approach to process twig queries. We try to reconstruct a twig query into a linear structural query automaton. To do so, we redesign the automata construction algorithm and propose a series of action transition to render the behavior of automata. The match processing algorithm is also modified to respond to the operations of action transitions. Moreover, a new evaluation approach of "//" operator is also proposed. Compared with the structure join way, the needed storage amount of intermediary results is decided by the number of *twigs* in our automata match method other than the length of the query expression. Additionally, because of the equivalence of automata and regular expressions, regular operators like closure need not to be rewritten any more and can be evaluated efficiently.

The rest of this paper is organized as follows. Section 2 gives some important definitions such as path schema tree and path expression. Section 3 describes the core data structure in our approach. Section 4 provides a detail statement of our new twig query algorithm. Section 5 shows experimental results. Our conclusions are contained in Section 6.

2 Preliminaries

2.1 XML Document and Path Schema Tree

In DOM [21] specification, an XML document is treated as a tree, and the entities of the document are nodes of the tree. Figure 1 gives an example of XML data tree. To concentrate on twig queries in this paper, only element nodes are concerned in XML documents. The numbers following "&" stand for IDs of the element nodes in database, and the words before IDs give the names of the nodes.

For an arbitrary node in an XML tree, the names of nodes from root to it construct a sequence, which is called path schema. For example, the path schema of node &12 in Fig. 1 is $a.d.e.f.j$. A structure named path schema tree is used to index path schemas of all nodes in an XML data tree. The corresponding path schema tree of data tree in Fig.1 is illustrated in Fig.2. The nodes in a path schema tree are called schema node and each schema node is related to some instance nodes in the XML data tree. Letters and numbers in the circles stand for the names and IDs of the schema nodes, and numbers in the bracket are IDs of the corresponding instance nodes. All the instance nodes of a schema node compose the extent of it, denoted by $Ext(s)$ for an instance node s. For two schema nodes s_1 and s_2, s_1 is defined as the parent node of s_2 if and only if there exist two instance nodes $i_1 \in Ext(s_1)$ and $i_2 \in Ext(s_2)$ that i_1 is the

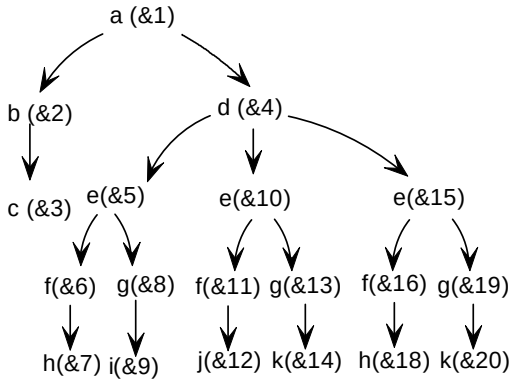

Figure 1: Sample XML data tree

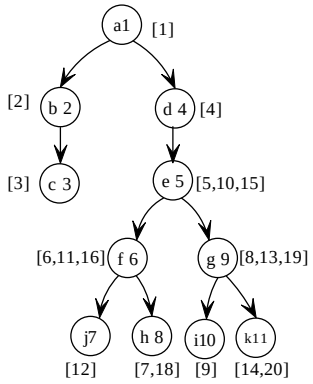

Figure 2: Sample path schema tree

```
AbstTwigExpr ::= '/' RltvTwigExpr
RltvTwigExpr ::= RltvTwigExpr BinaryOp RltvTwigExpr
               | RltvTwigExpr UnaryOp
               | Step
    BinaryOp ::= '/' | '//'
     UnaryOp ::= '*' | '+' | '?'
        Step ::= Name
               | Name '[' Predicate ']'
   Predicate ::= RltvTwigExpr
```

Figure 3: BNF syntax of twig path expression

parent node of i_2. In simple path expression queries, a path schema tree employs a role of the sentence as the input of automata. When an automaton runs with a path schema tree and stops at its accepting states, the result of the query lies in the extent of the current schema nodes. For example, the results of query $a/d/e/f/h$ on the document according to Fig.1 are &7 and &18. If there are predicates in a query expression, some of the candidate results in the extent are filtered out for not conforming to the predicates. Therefore, the final result of the query is a subset of the extent in this situation. The main purpose of this paper is to find real results when the predicates are path expressions themselves.

2.2 Twig Path Expression Queries

In XPath [19] specification, the predicates are quoted by brackets "[" and "]". If a predicate is a path expression other than a value testing, the predicate's value is true if the sub-query according to the testing node having results. For example in Fig.1, the result of query $/a/d/e[g/i]/f/h$ is &7. This paper concentrates on discussing twig queries, so we consider all predicates as path expressions but value testing expressions.

In Fig.3, a twig expression is a linear path expression if predicates are eliminated. This

linear path expression is called main path expression, while other expressions inside predicates are called predicate path expressions. Since predicates are used as filters, the result of a twig query is a subset of the result of its main path. Thus, the key work for evaluating a twig query is to find the appropriate ones within the results of its main path expression query.

2.3 Basic Automata Match Approach

Basic AM is an efficient approach to evaluate XML path queries. It supports all basic operators used in regular path expressions such as "$*$", "$+$", "?" and "$|$". In basic AM algorithm, all XML path expression queries are first converted into query automata as finite state automata. If the path schema of path instances in the XML data tree is accepted by the query automata, the corresponding path instances will be included in the result set. The algorithm matches path schema tree nodes with a set of automata states. It uses a bi-tuple $(Schema, Mark)$ to represent intermediary results, where $Schema$ is a set of path schema nodes and $Mark$ is a set of marks to indicate the real result. The instance set represented by $(Schema, Mark)$ is $Ext(Schema) \cap Mark$. AM approach uses marks to handle simple predicates.

3 Twig Automata

3.1 Query Automata

A deterministic finite automaton (DFA) is often denoted by "five-tuple" notation:

$$A = (Q, \Sigma, \delta, q_0, F)$$

where A is the name of the DFA, Q is its set of states, Σ its input symbols, δ its transition function, q_0 its start state, and F its set of accepting states. For an automaton A, if not exclaim, we use Q_A to denote its state set, and so on. We use automata to represent queries in XML query processing procedure. Since a query is represented by expressions, conversion methods from path expressions to automata should be found. First we come to the simplest situation. If a path expression contains no predicate and no "//" operator, it is certain a regular path expression. Conversion from this kind of expressions to automata can be done conforming to traditional automata theory [9]. Input symbol set Σ is the set of XML element names. There is also another name "empty" denoted by ε in Σ. It is used as the label of empty transition in the transformation from expression to automata. Since "//" is not a regular expression operator, it needs special representation and evaluation. Furthermore, common DFAs cannot represent twig queries exactly, so our definition of automata is changed slightly.

3.2 Slash-slash Extension of Automata

Slash-slash ("//") operator which is an abbreviation of self-and-descendant axis is a special operator in XPath. It is not a regular expression operator and expressions containing it cannot

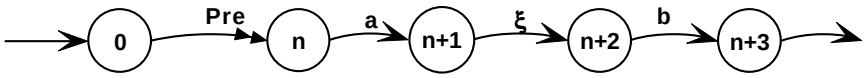

Figure 4: Sample query automaton with slash-slash extension

Name	Action
CONSTRUCT	Starts a twig session
PUSH	Stores schemas and marks in stacks
POP	Pop the stacks
RESTORE	Restore the schemas on top of schema stack
FILTERUP	Filter ancestor set with descendant set
FILTERDOWN	Filter descendant set with ancestor set

Figure 5: Action names of twig automata

be transformed into automata directly. There are several ways to compute it, such as query rewriting [17] and containment join [12, 22]. In this paper, we transform this operator into a particular name in order to evaluate it directly with automata match algorithm. A Greek letter ξ is used to denote a *wildcard* character, which can replace any possible expression. An expression with "//" is transformed into an expression containing ξ according to the equation:

$$E_1//E_2 \Leftrightarrow E_1/\xi/E_2$$

The correctness of this substitution is obvious. Figure 4 gives an example of query automata converted from $Pre/a//b/...$, where Pre is considered as some steps before a.

3.3 Twig Extension of Automata

We consider a path expression without predicates as a linear expression, which is exactly a sequence for automata to run with. When predicates with sub-expressions appear, the expression becomes a tree. We decompose the tree and reconstruct it into a linear expression by connecting the sub-paths with some special transitions. Labels on these transitions are some special names, each of which corresponds to a series of actions when they are read by automata. Their names and meaning are listed in Fig.5.

Here we introduce two stacks *sk_schema* and *sk_mark* to record the intermediary results. Take $a/d/e\,[g/i]/f/h$ for example, because query expression node e is the node with predicate, we call it key node. When an automaton reads a key node, there exist two paths of nodes to be read at the same time, i.e. g/i and f/h in the example. The path of predicate (g/i) is named predicate path and the other path is named main path. Considering the meaning of twig query expression, paths of predicate should be read first. However, for the purpose of computing the main path after computing the predicate one, the intermediary results of the key node must be recorded before reading the predicate path. These intermediary results are stored in *sk_schema*

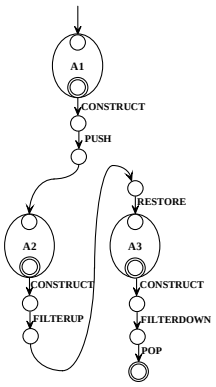

Figure 6: Construction procedure

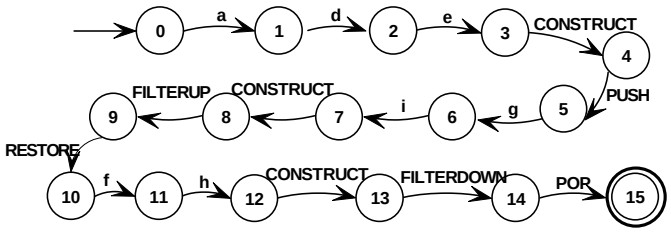

Figure 7: Sample twig query automaton

and *sk_mark*, path schema tree nodes in *sk_schema* and instance node marks in *sk_mark*. After computing predicate path (g/i), the automaton comes back to the key node (e), restores schema node set from *sk_schema* and marks from *sk_mark*, filters instance set with results generated from the predicate path, stores instance set again, and then computes the main path (f/h) with the schema node set just been restored. After the evaluation of the main path, the results (generated by f/h) should be filtered again with all the marks stored in *sk_mark*.

Action name `CONSTRUCT` starts a twig session. It generates current schema names and instance marks according to automata status. Name `PUSH` pushes current schema name set into *sk_schema* and instance marks into *sk_mark* while `POP` pops them out. Another name `RESTORE` restores the schema set on the top of *sk_schema*. Name `FILTERUP` and `FILTERDOWN` act on two instance set. One is ancestor set and the other is descendant set. `FILTERUP` filters out the instance in the ancestor set, any descendant of which is not in the descendant set. `FILTERDOWN` does exactly the contrary, which filters descendant set with ancestor set. We use Σ_{act} to denote the set of action names, thus the actual name set used by twig query automata is $\Sigma \cup \Sigma_{act}$. Since ξ sometimes behaves like action names, we also consider ξ as an action name and $\xi \in \Sigma_{act}$.

Now we come to the algorithm:

TWIG-TO-AUTO(n)

1 **switch**
2 **case** CHILD-COUNT(n) $= 0$:
3 **return** CONVERT(n)
4 **case** CHILD-COUNT(n) $= 1$:
5 **return** CONNECT(CONVERT(n), TWIG-TO-AUTO(FIRST-CHILD(n)))
6 **case** CHILD-COUNT(n) > 1 :
7 $A \leftarrow$ CONVERT-TO-AUTO(pet)
8 $n_1 \leftarrow$ NEW(), $n_2 \leftarrow$ NEW()
9 $Q_A \leftarrow Q_A \cup \{ n_1, n_2 \}$
10 $\delta_A \leftarrow \delta_A \cup \left(\bigcup_{f \in F_A} \{ (f, \texttt{CONSTRUCT}, n_1) \} \right) \cup \{ (n_1, \texttt{PUSH}, n_2) \}$

11 $F_A \leftarrow \{ n_2 \}$
12 $c \leftarrow$ First-Child
13 $M \leftarrow$ Twig-To-Auto(c)
14 $c \leftarrow$ Next-Sibling(c)
15 **while** $c \neq$ nil
16 **do** $A \leftarrow$ Connect$(A,$ Twig-To-Auto$(c))$
17 $n_1 \leftarrow$ New$(), n_2 \leftarrow$ New$(), n_3 \leftarrow$ New$()$
18 $Q_A \leftarrow Q_A \cup \{ n_1, n_2\, n_3 \}$
19 $\delta_A \leftarrow \delta_A \cup \left(\bigcup_{f \in F_A} \{ (f, \texttt{CONSTRUCT}, n_1) \} \right)$
20 $\delta_A \leftarrow \delta_A \cup \{ (n_1, \texttt{FILTERUP}, n_2), (n_2, \texttt{RESTORE}, n_3) \}$
21 $F_A \leftarrow \{ n_3 \}$
22 $c \leftarrow$ Next-Sibling(c)
23 $A \leftarrow$ Connect(A, M)
24 $n_1 \leftarrow$ New$(), n_2 \leftarrow$ New$(), n_3 \leftarrow$ New$()$
25 $Q_A \leftarrow Q_A \cup \{ n_1, n_2\, n_3 \}$
26 $\delta_A \leftarrow \delta_A \cup \left(\bigcup_{f \in F_A} \{ (f, \texttt{CONSTRUCT}, n_1) \} \right)$
27 $\delta_A \leftarrow \delta_A \cup \{ (n_1, \texttt{FILTERDOWN}, n_2), (n_2, \texttt{POP}, n_3) \}$
28 $F_A \leftarrow \{ n_3 \}$
29 **return** A

We describe the algorithm using the example. First we divide path expression $a/d/e\,[g/i]/f/h$ into three parts: root-end main path $a/d/e$, predicate path g/i and leaf-end main path f/h. And then we transform them into an automata using classical methods. We mark the three generated automata as A_1, A_2 and A_3 as seen in Fig.6. For the sake of clarity, we ignore all these automata states except start and accepting states. Then we add action transactions and some states and link them together like Fig.6. The New method creates and returns a new automata state, Convert transforms a regular expression fragment into an automaton, and Connect connects two automata to form a new one. Finally, the generated automata are normalized. The query automata converted from the sample query is shown in Fig.7. If predicate path or leaf-end main path itself is a twig path as well, the procedure above can be invoked recursively.

4 Twig Automata Match Algorithm

4.1 Computation of Slash-slash

In TAM approach, we calculate "//" operator directly during the running of automata instead of rewriting it. We have described how to transform "//" operator into an action name ξ. In this section we introduce how to process ξ and leave other action names to Section 4.2. We use

a relative short algorithm to compute it:

FIND-DESCENDANT(S, N)

```
1  L ← S, R ← ∅
2  while L ≠ ∅
3      do s ← REMOVEFIRST(L)
4          L ← L ∪ CHILDREN(s)
5          for  each c in CHILDREN(s)
6              do if NAME(c) ∈ N
7                     then R ← R ∪ { c }
8  return R
```

The algorithm is used to find all descendants of a set of schema nodes S whose name is in the name set N, and returns them (R). In the algorithm, W is a wait list to help to traverse the sub path schema tree, REMOVEFIRST removes the first element of a list and returns it, CHILDREN returns all children nodes of a schema node and NAME its name. Like basic AM, TAM uses schema nodes and marks to represent instance results, thus we only need to find the descendant schema nodes in path schema tree rather than traverse the XML data tree. Because path schema tree is much smaller than XML data tree, the performance of this algorithm is expected.

4.2 Operation of Action Transitions

We will give the detailed operation of action transactions in this section. These transitions are additionally added to the query automata to support twig query processing, so they need additional structures and process procedure. We use two stacks sk_schema and sk_mark to record intermediary results, which correspond to $Schema$ and $Mark$ of the result representation respectively. The reason of using stack is to cope with the situation when twigs are nested, like $a/b[c[d]/e]/f$, whose twig predicate is a path expression with twig itself. Each stack has three basic methods PUSH, POP and TOP. Now we give the algorithm:

PERFORM-ACTION(a)

```
1  switch
2    case a = CONSTRUCT :
3          CALC-MARK(Mark)
4    case a = PUSH :
5          PUSH(sk_schema, L)
6          PUSH(sk_mark, Mark)
7    case a = POP :
8          POP(sk_schema)
9          POP(sk_mark)
```

```
10    case a = RESTORE :
11          L ← Top(sk_schema)
12    case a = FILTERUP :
13          T ← Top(sk_mark)
14          Pop(sk_mark)
15          Filter(Mark, T)
16          Push(sk_mark, Mark)
17    case a = FILTERDOWN :
18          T ← Top(sk_mark)
19          Filter(T, Mark)
```

In the algorithm, a is the action name, L is a global variable storing all the schema nodes needed to be processed in TAM algorithm, and Filter is the method to filter the first set with the second which can be realized using join or merge methods.

4.3 Detailed Twig Automata Match Algorithm

In this section, we discuss TAM algorithm in detail. The general procedure of TAM algorithm and basic AM algorithm are alike: construct a query automata from query expression, match each path schema tree node with a query automata state, mark the exceptional results when processing predicates, and finally get the results when the query automaton arrives its accepting states. The difference is the evaluation of action transitions. The detailed twig automaton match algorithm is given below as Twig-Automata-Match.

Twig-Automata-Match($root, A$)

```
1  match[root] ← q0A, L ← { root }
2  while L ≠ ∅
3     do s ← RemoveFirst(L), q ← match[s]
4        if q ∈ FA
5          then return
6        for  each a, δA(q, a) ≠ NIL
7           do switch
8                 case a = ξ :
9                       q ← δA(q, a)
10                      R ← Find-Descendant({ s }, ⋃_{δA(q,a)≠NIL} a)
11                      for  each r in R
12                         do match[r] ← δA(q, a)
13                      L ← L ∪ R
14                case a ∈ Σact :
15                      Perform-Action(a)
```

```
16                    match[s] ← δ_A(q, a)
17                    L ← L ∪ { s }
18              case a ∈ Σ :
19                    for each c in CHILDREN(s)
20                        do if NAME(c) = a
21                              then match[c] = δ_A(q, a)
22                                   L ← L ∪ { c }
```

In the algorithm, *root* is the root node of path schema tree, A is the twig query automaton, and L is the same global variable introduced in Section 4.2. Method CONSTRUCTRESULT constructs instances using a schema node and current marks and returns them.

Now we will demonstrate the running procedure with an example. Figure 8 shows the content of variables and some data structures while matching the path schema tree in Fig.2 and the automaton in Fig.7. It lists the changes of the name that the automaton is reading, the current automaton state, the content of wait list variable L, variable $Mark$, and the top element of two stacks *sk_schema* and *sk_mark*. We will explain some important steps in the processing. From step 1 to 4, the root-end main path is matched, key node e is read, and the automaton reaches state 3. When `CONSTRUCT` is read at step 5, a mark is generated from current path schema tree node 5, and it is pushed into stack together with L at step 6. At step 7 and 8, predicate path is matched, and a mark {&9} is got at step 9. At step 10, `FILTERUP` is read and the top element of *sk_mark* is filtered by $Mark$. Because neither &10 nor &15 is the descendant of &9, only &5 is left in the top stack element. Schema wait list is restored from stack *sk_schema* into L at step 11, and the leaf-end main path is matched during step 12 and 13. And then at step 15, `FILTERDOWN` is read, $Mark$ is filtered by the top element of *sk_mark*, thus only &7 which is one of the descendants of &5 survives. Finally at step 16, both stacks pop their elements, and at the same time the automaton reaches state 15 which is its accepting state. Therefore, the final result which is constructed by L and $Mark$ is {&7}. The algorithm successfully stops.

4.4 Cost Analysis

In this section, we analyze the performance of TAM by comparing two methods: twig automata match algorithm (TAM) and structural join algorithm (STJ).

The CPU time of TAM is mainly made up of two parts: the time to match query automata and path schema tree and the time to filter the candidate results. Because path schema tree size is a very small index, match operation is relatively fast. Therefore, filter operation occupies most of the CPU time when the data size is very large. For a fixed twig query, there are two main factors that affect the CPU time:

1. the number of filter operation;

2. the size of two sets to be processed.

Thus, in order to compare the former factor's effect on these two methods conveniently, we assume the time of a filter operation equals to the time of a STJ Join which is denoted by T_j. Twig query's CPU time is only related to the number of filter operations and automata match operations with schema tree. We assume that the average time cost by automata match operation is T_a per node. The CPU time of TAM and STJ can be evaluated as:

$$T_{TAM} = (N_{leaf} - 1) * T_j + N_{key} * T_j + N_{schema} * T_a$$
$$T_{STJ} = (N_{total} - 1) * T_j$$
$$N_{total} = N_{leaf} + N_{key} + N_{key'} + N_{pre}$$

where N_{leaf} is the number of leaf nodes in a query expression tree, N_{key} is the number of key nodes in the main path, N_{total} is the total node number in query expression tree, and N_{schema} is the total node number in path schema tree. Take query expression $/a/d/e[g/i]/f/h$ for example, the total node number is 7, so the number of join operations in STJ is 7-1=6. The main path is $a/d/e/f/h$, in which the key node is e, the number of key node is 1, and the leaf number is 2, therefore TAM does filter operation only twice.

$N_{key'}$ represents the node number in the main path whose degree is 1, while N_{pre} represents the node number in the predicate path whose degree is no less than 1. It can be drawn from the formula that: given a query expression Q, TAM does $N_{key'} + N_{pre}$ less times join operations

Step	**Name**	**States**	**L**	**Mark**	**sk_schema top**	**sk_mark top**
1		0				
2	a	1	{1}			
3	d	2	{4}			
4	e	3	{5}			
5	CONSTRUCT	4	{5}	{&5,&10,&15}		
6	PUSH	5	{5}	{&5,&10,&15}	{5}	{&5,&10,&15}
7	g	6	{9}		{5}	{&5,&10,&15}
8	i	7	{10}		{5}	{&5,&10,&15}
9	CONSTRUCT	8	{10}	{&9}	{5}	{&5,&10,&15}
10	FILTERUP	9	{10}	{&9}	{5}	{&5}
11	RESTORE	10	{5}	{&9}	{5}	{&5}
12	f	11	{6}		{5}	{&5}
13	h	12	{8}		{5}	{&5}
14	CONSTRUCT	13	{8}	{&7,&8}	{5}	{&5}
15	FILTERDOWN	14	{8}	{&7}	{5}	{&5}
16	POP	15	{8}	{&7}		

Figure 8: Running example of twig automata match algorithm

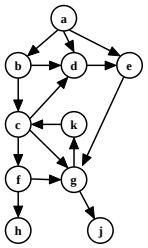

Figure 9: DTD of experiment document

Query	Expression
Q_1	$a/b/c[f/h \text{ and } g/j]/d$
Q_2	$a/b[//f]//e//j$
Q_3	$a/b/c/d/e/g[k/c]/j$
Q_4	$a/b//c[d/e]/f$

Figure 10: Query expressions for experiment

than STJ. In the worst case, $N_{key'} + N_{pre} = 0$ (like a[b]/c and a[b]/c[d]/e, etc.). Since the CPU time of TAM includes $N_{schema} * T_a$, in this case, TAM will be a bit slower than STJ. We use $(N_{key'} + N_{pre})/N_{total}$ to express the advantage of join number between TAM and STJ. The larger the ratio, the better TAM outperforms STJ.

The analysis above assumes the time cost by TAM filter operation is the same as the join operation in STJ. However, TAM and STJ use different size of data set when doing the operations. TAM uses instance set corresponding to a schema node and STJ uses element extent as their data set. The former is certainly not greater than the latter. Therefore, TAM will have better performance even if they have the same number of join operations.

5 Performance Evaluation

The experiment platform is a PIII 933HZ CPU, 128M of memory and a 30G hard disk computer. The operating system is Windows2000. We employ a native XML DBMS (XBase [14]) which is based on an OODBMS Fish [24, 23] and is conformed to ODMG [4] C++ binding. The programs are written on Microsoft Visual C++ 6.0.

To test the performance of automata match methods on twig queries, we implemented two query methods in the same platform: automata match over twig queries method (TAM) and an Improved Structural Join method — stack tree join(STJ). STJ decomposes a twig path expression into steps and then joins the query results of all these step to get the final results.

We adopt IBM XML Generator to generate XML documents to test the query performance on complex structure data set. We generate a data set (TMark) having five different documents (5M, 10M, 20M, 40M and 80M), through customizing some document parameters such as the maximal element occurrence and document depth. These documents conform to the DTD shown in Fig.9. The 80M XML document contains about two million elements. We have designed four query expressions (Q_1–Q_4) to test the effect to query performance of path length, results number and document size.

According to the experiment results, we can see that TAM outperforms STJ significantly in all aspects shown above. The reason is that for each join operation of STJ, STJ accesses the instances in database. While in TAM method, the number of instances accessed is far less than that of STJ. Take Q_3 for example, except for the access when getting the result, there are

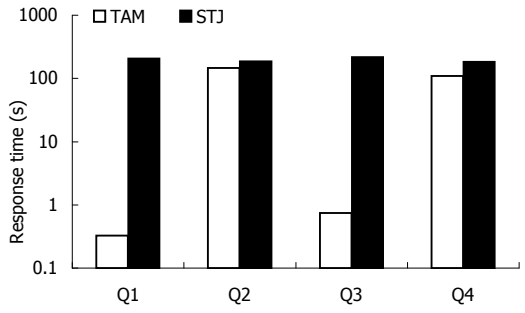

Figure 11: Performance on 80M document

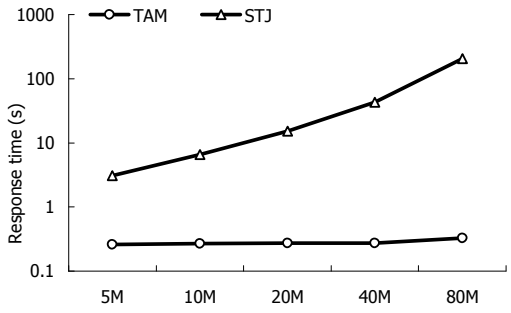

Figure 12: Long path (Q_1) performance

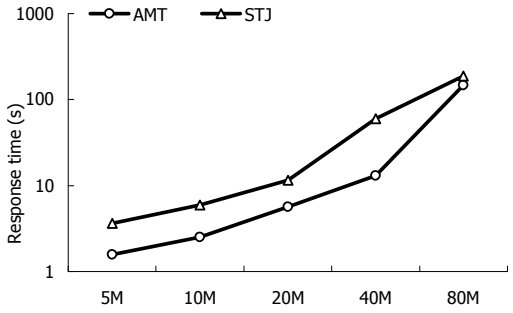

Figure 13: Short path (Q_2) performance

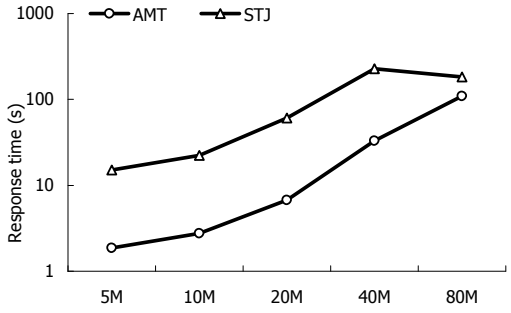

Figure 14: Common path (Q_4) performance

only two join operations in middle-operation and three sets of nodes are accessed: g satisfies $a/b/c/d/e/g$, c satisfies $a/b/c/d/e/g/k/c$ and j satisfy $a/b/c/d/e/g/j$. We can see clearly that in TAM, the number of accessed nodes is only relative to the twig number. In STJ, nevertheless, the number of accessed nodes is in direct proportion to the path length. It is obvious that the number of accessed nodes of TAM is well under that of STJ. In Q_1–Q_4, the TAM advantage frequency $(N_{key'} + N_{pre})/N_{total}$ are 1/2, 2/5, 2/3, 1/2 respectively. Therefore, if only take the number of join operation into account, Q_3 is the best, Q_1 and Q_4 take the second places, and Q_2 is the worst. Because schema tree has better filter effect on Q_1 than Q_3, the performance of Q_1 is better than Q_3. Figure 12 shows the query performance of Q_1 on different size of documents where Q_1 a long path query expression. In this type of queries, the final result number is relatively small compared with the whole XML document, thus document's size has little effect on the query time of Q_1. TAM method keep the overwhelming advantage on long path expression queries[15], faster than other methods such as STJ from several to decades times. Figure 13 shows the performance of Q_2. For Q_2, we mainly tend to test the performance on short path expressions and computing slash-slash operator through schema tree. In this experiment, the performance of TAM is about twice of STJ. Because this query has a great deal of results (about tenth of the whole document elements), query time increases proportionately with the growth of document size.

6 Conclusions

In this paper, we propose a new automaton match approach for twig path query, TAM. This method extends query automata by adding some action transitions to automaton to order automata to do special operations while running. TAM evaluates twig queries by transfering twig query expressions to query automata with action transitions and then matches path schema tree with it. We also design another algorithm to resolve the containment question in XPath, based on traversal method on path schema tree. Extensive experiment results show that our new TAM approach performs remarkably well on XML twig queries.

References

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In ICDE02 [10].

[2] M. Altmel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 53–63, Cario, Egypt, September 10–14 2000. Morgan Kaufmann.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In Franklin et al. [8].

[4] R. Cattell, D. Barry, M. Berler, et al. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann Publishers, Inc, San Francisco, California, 2000.

[5] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. Ng, and D. Srivastava. Counting twig matches in a tree. In *Proceedings of IEEE International Conference on Data Engineering*, pages 595–604, Heidelberg, Germany, April 2001.

[6] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In P. A. Bernstein, Y. E. Ioannidis, R. Ramakrishnan, and D. Papadias, editors, *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 263–274, Hong Kong SAR, China, August 20–23 2002.

[7] Y. Diao, P. Fischer, M. Franklin, and R. To. YFilter: efficient and scalable filtering of XML documents. In ICDE02 [10].

[8] M. J. Franklin, B. Moon, and A. Ailamaki, editors. *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, USA, June 3–6 2002.

[9] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 2001.

[10] *Proceedings of the the 2002 International Conference on Data Engineering*, San Jose, California, USA, February 26–March 1 2002.

[11] *Proceedings of the the 2003 International Conference on Data Engineering*, 2003.

[12] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML data for efficient stuctural join. In ICDE03 [11].

[13] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed XML documents. In J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, editors, *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 273–284, Berlin, Germany, September 9–12 2003.

[14] H. Lu, G. Wang, G. Yu, Y. Bao, J. Lv, and Y. Yu. XBase: Making your gigabyte disk queriable. In Franklin et al. [8].

[15] J. Lv, G. Wang, J. X. Yu, G. Yu, H. Lu, and B. Sun. A new path expression computing approach for XML data. In *Proceedings of 1st VLDB Workshop on Efficiency and Effectiveness of XML Tools and Techniques (LNCS 2590)*, Hong Kong SAR, China, August 2002.

[16] J. Lv, G. Wang, J. X. Yu, G. Yu, H. Lu, and B. Sun. Performamce evaluation of a DOM-based XML database: Storage, indexing and query optimization. In *Proceedings of WAIM 2002 (LNCS 2419)*, Beijing, China, August 2002.

[17] B. Sun, J. Lv, G. Wang, and G. Yu. Efficient evaluation of XML path queries with automata. In *Proceedings of WAIM 2003 (LNCS 2762)*, Chengdu, China, August 2003.

[18] W3C Recommendation. *Extensible Markup Language (XML) 1.0 (Second Edition)*, October 6 2000. http://www.w3.org/TR/REC-xml.

[19] W3C Working Draft. *XML Path Languages (XPath), ver 2.0*, December 20 2001. Tech. Report WD-xpath20-20011220, W3C, 2001, http://www.w3.org/TR/WD-xpath20-20011220.

[20] W3C working draft. *XQuery 1.0: An XML Query Language*, June 2001. Technical Report WD-xquery-20010607, World Wide Web Consortium.

[21] W3C Working Draft. *Document Object Model (DOM) Level 3 Core Specification*, 2003. http://www.w3.org/TR/DOM-Level-3-Core.

[22] W. Wang, H. Jiang, H. Lu, and J. X. Yu. PBiTree coding and efficient processing of containment join. In ICDE03 [11].

[23] G. Yu, K. Kaneko, G. Bai, and A. Makinouchi. Transaction management for a distributed store system WAKASHI design, implementation and performance. In *Proceedings of the 12th International Conference on Data Engineering*, pages 460–468. New Orleans: IEEE Computer Society, 1996.

[24] G. Yu, G. Wang, and A. Makinouchi. A distributed and parallel object database server system for Windows NT. In *Proceedings of Conference on Software: Theory and Practice*, Beijing, China, August 2000.

[25] J. X. Yu, G. Wang, H. Lu, G. Yu, J. Lv, and B. Sun. Automata match: A new XML query processing approach. *Special Issue on Trends in XML Technology for the Global Information Infrastructure, International Journal of Computer Systems, Science and Engineering*, 2003.